

Приложения на сортиращите алгоритми

Тодор Дуков

Обща информация за някои алгоритми за сортиране

Понякога се оказва много удобно да сортираме входните данни, защото това ни носи полезна информация. Затова е хубаво да се знаят различните алгоритми за сортиране и в какво те са добри. Нека ги сравним по тяхната сложност, като:

- $T_{avg}(n)$ е сложността по време в средния случай
- $M_{avg}(n)$ е сложността по памет в средния случай
- $T_{worst}(n)$ е сложността по време в най-лошия случай
- $M_{worst}(n)$ е сложността по памет в най-лошия случай

име	$T_{avg}(n)$	$M_{avg}(n)$	$T_{worst}(n)$	$M_{worst}(n)$
пирамидално сортиране	$\Theta(n \log(n))$	$\Theta(1)$	$\Theta(n \log(n))$	$\Theta(1)$
сортиране чрез сливане	$\Theta(n \log(n))$	$\Theta(n)$	$\Theta(n \log(n))$	$\Theta(n)$
бързо сортиране	$\Theta(n \log(n))$	$\Theta(\log(n))$	$\Theta(n^2)$	$\Theta(n)$

Въпреки че алгоритъмът за бързо сортиране е по-бавен в най-лошия случай от пирамидалното сортиране и сортирането чрез сливане, практически се оказва, че се справя по-добре. Разбира се, другите сортировки също имат предимства. Пирамидалното сортиране заема малко памет, което е много полезно във вградени системи. Сортирането чрез сливане се оказва по-бързо, когато данните са много големи и се съхраняват във външни устройства (да кажем, на твърд диск). Освен тези алгоритми има и други хубави алгоритми като сортиране чрез броене, но за съжаление ние няма да ги разглеждаме тук.

Два алгоритъма

Ще започнем с два често срещани алгоритъма, които се възползват от това, че данните идват сортирани:

- двоично търсене
- алгоритъма за задачата 2-sum

Нека започнем с алгоритъма за двоично търсене:

```
1 int binary_search(int *arr, int n, int val)
2 {
3     int left = 0, right = n - 1;
4
5     while (left <= right)
6     {
7         int mid = left + (right - left) / 2;
8
9         if (arr[mid] == val)
10            return mid;
11        else if (arr[mid] < val)
12            left = mid + 1;
13        else
14            right = mid - 1;
15    }
16
17    return -1;
18 }
```

При подаден сортиран масив от числа `arr` с размер `n` и стойност `val`, функцията `binary_search(arr, n, val)` ще върне индекс на `arr`, в който се намира `val`, ако има такъв, иначе ще върне `-1`:

Инварианта. При всяко достигане на проверката за край на цикъла на ред 5 имаме, че стойността `val` не се намира измежду двата масива `arr[0...left - 1]` и `arr[right + 1...n - 1]`.

База. При първото достигане имаме, че `left = 0` и `right = n - 1`. Наистина стойността `val` не се намира измежду двата масива `arr[0...0 - 1]` и `arr[n - 1 + 1...n - 1]`.

Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава `val` не се намира измежду `arr[0...left - 1]` и `arr[right + 1...n - 1]`, и понеже достигането е непоследно, $left \leq right$. Тогава $mid = \left\lfloor \frac{left + right}{2} \right\rfloor$, откъдето $left \leq mid \leq right$. Трябва да разгледаме следните три случая:

1 сл. `arr[mid] = val` – това няма как да е изпълнено понеже достигането е нефинално

2 сл. `arr[mid] < val` – понеже `arr` е сортиран, няма как `val` да се намира измежду `arr[0...mid]`, откъдето `val` не се намира измежду `arr[0... $\underbrace{mid + 1}_{left_{new}} - 1]$` и `arr[right + 1...n - 1]`.

3 сл. `arr[mid] > val` – напълно дуален на 2 сл.

Терминация. От цикъла винаги ще излезем, защото или ще открием `val`, или `right - left` ще намалява, докато не стане отрицателна. Излизането става по два начина:

- не е изпълнено условието на ред 5 т.е. $left > right$ – тогава $left - 1 \geq right$ и понеже `val` не се намира измежду `arr[0...left - 1]` и `arr[right + 1...n - 1]`, `val` не се намира във `arr[0...n - 1]`. Накрая алгоритъмът ще върне `-1`, което наистина е желания резултат.
- изпълнено е условието на ред 9 т.е. `arr[mid] = val` – тогава алгоритъмът коректно връща `mid`.

Нека сега разгледаме алгоритъма за задачата 2-sum:

```
1 bool two_sum(int *arr, int n, int target)
2 {
3     int left = 0, right = n - 1;
4
5     while (left < right)
6     {
7         if (arr[left] + arr[right] == target)
8             return true;
9         else if (arr[left] + arr[right] < target)
10            ++left;
11        else
12            --right;
13    }
14
15    return false;
16 }
```

Ще покажем, че при подаден сортиран масив `arr` с размер `n` и число `target`, функцията `two_sum(arr, n, target)` разпознава дали има $0 \leq i < j \leq n - 1$, за които:

$$arr[i] + arr[j] = target \quad // \text{ всяка такава двойка } (i, j) \text{ ще наричаме } \textit{диада}$$

Инварианта. При всяко достигане на проверката за край на цикъла на ред 5, всички диади са в `arr[left...right]`.

База. При първото достигане имаме, че `left = 0` и `right = n - 1`. Тогава твърдението е тривиално изпълнено.

Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава всички диади са в `arr[left...right]` и `left < right`. Разглеждаме три случая:

1 сл. `arr[left] + arr[right] = target` – това няма как да е изпълнено понеже достигането е нефинално

2 сл. `arr[left] + arr[right] < target` – тогава понеже `arr` е сортиран, за всяко $left + 1 \leq i \leq right$ имаме, че $arr[left] + arr[i] \leq arr[left] + arr[right] < target$. Това означава, че `left` не участва в никоя диада във `arr[left...right]`. Така получаваме, че всички диади се намират в `arr[$\underbrace{left + 1}_{left_{new}}...right]$`

3 сл. `arr[left] + arr[right] > target` – напълно дуален на 2 сл.

Терминация. От цикъла винаги ще излезем, защото или ще открием `val`, или `right - left` ще намалява, докато не стане 0. Излизането става по два начина:

- не е изпълнено условието на ред 5 т.е. `left ≥ right` – тогава няма диади в `arr`, и алгоритъмът коректно ще върне `false`.
- изпълнено е условието на ред 9 т.е. `arr[left] + arr[right] = target` – тогава алгоритъмът връща `true` т.е. точно това, което искаме.

Двата алгоритъма са доста подобни, използват една често срещана техника за търсене в дадено множество от елементи. Търсенето започва с цялото множество и то постепенно се смялява. Разбира се, тук разгледаните алгоритми имат разлика в сложността, поради разликата в стесняването:

- първият алгоритъм има сложност $O(\log(n))$ понеже винаги разликата между `left` и `right` намалява двойно
- вторият алгоритъм има сложност $O(n)$ понеже винаги разликата между `left` и `right` намалява с единица

Задачи

Задача 1. Да се напише колкото се може по-бърз алгоритъм `find_peak(arr, n)`, който приема масив от различни числа `arr` с размер $n \geq 3$, за който има $0 < i < n - 1$ такава, че `arr[0...i]` е сортиран възходящо и `arr[i...n-1]` е сортиран низходящо, и връща това `i`. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 2. Да се напише колкото се може по-бърз алгоритъм `search_range(arr, n, val)`, който при подаден сортиран масив `arr` с размер `n` и число `val`, което се намира в масива, връща най-малкият и най-големият индекс, на който `val` се намира в `arr`. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 3. Да се напише колкото се може по-бърз алгоритъм `k_sum(arr, n, k, target)`, който при подаден сортиран масив `arr` с размер `n` и числа $k \geq 2$ и `target`, връща дали има $0 \leq i_1 < i_2 < \dots < i_k \leq n - 1$, за които:

$$\text{arr}[i_1] + \text{arr}[i_2] + \dots + \text{arr}[i_k] = \text{target}$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4. Да се напише колкото се може по-бърз алгоритъм `k_closest(arr, n, k, target)`, който принтира най-близките `k` числа в `arr` до `target`. След това да се докаже неговата коректност, и да се изследва сложността му по време. Може ли да се напише по-бърз алгоритъм при предположение че `arr` е сортиран?

Задача 5. Да се напише колкото се може по-бърз алгоритъм `sort(arr, n)`, който приема масив `arr` с размер `n`, съставен от числата 0, 1 и 2, и го сортира. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6. Да се напише колкото се може по-бърз алгоритъм `merge_intervals(intervals, n)`, който приема масив `intervals` с размер `n`, съставен от двойки числа, които представят някакъв затворен интервал, и връща нов масив, в който са сляти всички интервали с непразно сечение. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 7. Да се напише колкото се може по-бърз алгоритъм `sorted_squares(arr, n)`, който приема масив `arr` с размер `n`, и връща нов масив от квадратите на `arr`, който е сортиран. След това да се докаже неговата коректност, и да се изследва сложността му по време.